Title: A “yo-yo” parsing algorithm for a large class of van Wijngaarden grammars

Running title: Parsing van Wijngaarden grammars

Author: Anthony J. Fisher

Address: Department of Computer Science, The University of York, York YO1 5DD, U.K.

Summary

An algorithm is described for parsing a large class of van Wijngaarden grammars in polynomial time. The algorithm is based on Earley's context-free parsing algorithm, with the addition of a bottom-up component and a mechanism for implementing consistent substitution. In contrast to earlier algorithms, this algorithm works for grammars which are both non-left-bound and non-right-bound.

1. Introduction

The van Wijngaarden grammar (VWG) is perhaps the most elegant means known of describing non-context-free languages. In spite of this, there have been few attempts to use VWGs in syntax-driven programs, such as compilers for extensible languages, macro-processors, or syntax-checkers.

The parsing problem for unrestricted VWGs is unsolvable [10]. VWG parsing algorithms therefore work only with restricted subsets of VWGs. Previous algorithms, of which only three are known, enforce restrictions on the grammar which are so severe that it is difficult to use these algorithms in practical applications. In particular, two of these algorithms [11, 5] are based on Deussen's [1] classification of (strictly) left-bound and right-bound rules. (These terms will be defined later.) Informally, a VWG is parsable by a top-down algorithm if all of its rules are right-bound, and by a bottom-up algorithm if all of its rules are left-bound. Unfortunately, these restrictions are so constraining in practice that parsing methods based on them can be used only for "toy" grammars. The third algorithm [4] has restrictions which are even more constraining. These observations (or perceptions) largely account for the dearth of applications of the various parsing algorithms.

By contrast, the new algorithm described in this paper allows both non-left-bound and non-right-bound rules. The parser, which is based on Earley's context-free parsing algorithm [3], employs both top-down and bottom-up techniques as appropriate; i.e. it uses "mixed strategy" parsing, sometimes called *yo-yo parsing*.

2. Background and preliminary definitions

We shall use some results from formal language theory. These are given in, for example, Salomaa [8].

VWGs are defined in the revised report on Algol 68 [13], which we shall refer to as "the Report". Pagan [7] gives an excellent informal introduction. The following definition, which is more concise than that of the Report, is that used by Fisher [5], which is in turn adapted from Wegner's definition [11]. As usual, let $\varnothing$ denote the empty set, and let λ denote the empty word.

Definitions.

A *van Wijngaarden grammar* (VWG) is an ordered 7-tuple $G = (M, V, N, T, R_M, R_V, S)$, where

M is a finite set of *metanotions*;

V is a finite set of *small syntactic marks*, $M \cap V = \varnothing$;

N is a set of *hypernotions*, a finite subset of $(M \cup V)^+$;

T is a finite set of *terminals*;

R_M is a finite set of *metarules* $X \to Y$, where $X \in M$, $Y \in (M \cup V)^*$, such that for all $m \in M$, (M, V, m, R_M) is a context-free grammar;

R_V is the finite set of *hyperrules* $h_0 \to h_1, h_2, ..., h_m$, where $h_0 \in N$ and $h_i \in (T \cup N \cup \{ \lambda \})$ for $1 \le i \le m$;

$S \in N$ is the *starting symbol*.

R_M is called the *metasyntax* of G, and R_V is called the *hypersyntax* of G.

Given a VWG $G = (M, V, N, T, R_M, R_V, S)$, we define the set R_S of *strict rules* of a hyperrule $r = h_0 \to h_1, h_2, ..., h_n$ containing the $n \ge 0$ metanotions $m_1, m_2, ..., m_n$ as follows:

$$R_S(r) = \{\ \phi(h_0) \to \phi(h_1), \phi(h_2), \ldots, \phi(h_m)\ \mid$$

ϕ is a homomorphism with:

$\phi(v) = v$ for $v \in V$ or $v = \lambda$,

$\phi(m_i) \in L((M, V, m_i, R_M))$,

$\phi(h_0) \neq \lambda\ \}$.

ϕ is called a *consistent substitution*. ϕ, applied to a hypernotion, yields a word over V, which is free of metanotions. Words over V are called *protonotions*. A hyperrule $r = h_0 \to h_1, h_2, \ldots, h_n$ is *metanotion-free* iff h_i is a protonotion for each $i = 0, \ldots, n$.

Let N_S denote the set of *strict notions*, defined by

$$N_S = \{\ \phi(h)\ \mid\ \phi \text{ is a consistent substitution and } h \in N\ \}.$$

The *language generated* by G, $L(G)$, is the language generated by the set of strict rules, considered as a possibly infinite context-free grammar:

$$L(G) = \{\ x \in T^*\ \mid\ S \Rightarrow_G{}^* x\ \},$$

where the relation $\Rightarrow_G$ is defined by

$X \Rightarrow_G Y$ iff $\exists\ P, Q \in (N_S \cup T)^*$ such that

$X = P\,X'\,Q$ and $Y = P\,Y'\,Q$ and

$X' \to Y' \in R_S(r)$ for some $r \in R_V$.

The set of strict rules which are applied in a given derivation is called a *strict syntax* of G. It is generally stipulated that, for all derivations of finite sentences, the strict syntax is finite. We shall assume that this is the case. See Sect. 0.4.2 of the Report for a short discussion on this point.

The following definitions are due to Deussen [1]. Let $hr = l \to r \in R_V$ be a hyperrule, and let $|x|_m$ denote the number of occurrences of the metanotion m in x. hr is said to be:

(*i*) *right-bound* iff $\forall\ m \in M,\ |r|_m \neq 0 \Rightarrow |l|_m \neq 0$; and

(*ii*) *left-bound* iff $\forall\ m \in M,\ |l|_m \neq 0 \Rightarrow |r|_m \neq 0$.

The grammar G is said to be right- (resp. left-) bound iff, $\forall\ hr \in R_V$, hr is right- (resp. left-) bound.

Notation.

We shall use the notation of the Report.

The symbol "," of the definition, which separates hypernotions on the right-hand side of a hyperrule, is written as a comma. Hyperrules are distinguished from metarules by the separator used for $\rightarrow$, which is ":" in a hyperrule and "::" in a metarule. The symbol ";" means "or", and is used in hyperrules and metarules to combine rules with identical left-hand sides, with the same meaning as "|" in Backus-Naur form.

Metanotions are written as upper-case words. The alphabet V of small syntactic marks is { a, b, ..., z, <, > }.

A terminal is written as a protonotion which ends in "symbol". The VWG is supposed to be augmented by what the Report calls "representation rules", which define a one-to-one correspondence between "symbol" protonotions (e.g. "plus symbol") and "real" terminals (e.g. "+"). From a formal point of view, symbol protonotions can be viewed as ordinary protonotions, and the representation rules can be viewed as ordinary hyperrules whose right-hand side is a single terminal. Consequently, and without loss of generality, we assume that terminals in the VWG occur only on the right-hand side of hyperrules of the form l : t where l is a protonotion.

3. Previous work

The problem of parsing VWGs has not attracted as much attention as one might have expected. Some effort seems to have been diverted from a study of "pure" VWGs, as defined by the Report, to less "pure" but more tractable variants. There are many such variants, of which affix grammars are perhaps the best known. An interesting and detailed account of VWG variants is given by Simonet [9], whose thesis contains a useful bibliography.

It is clear that there can be no algorithm for parsing all "pure" VWGs, since VWGs are far too powerful; as powerful, in fact, as Chomsky type 0 grammars [10]. There have been three previous attempts to find an algorithm for parsing a useful subset of VWGs.

An early attempt by Fisher [4] was to try to transform a VWG into a context-free grammar which generates "approximately" the same language as the given VWG, which one can think of as an infinite CFG. This approach worked fairly well for small VWGs, but the size of the transformed grammar increased very rapidly with the size of the input grammar. Also, the transforming algorithm was not guaranteed to terminate unless an arbitrary limit were imposed on the depth of recursion. However, a working parser, and its associated table-generator program, were implemented. This method cannot, in fairness, be said to work well with grammars of any useful size.

Wegner [11] also considers a VWG as a finite specification of an infinite CFG. He shows how, given a VWG G which satisfies certain restrictions, one can obtain a CFG G' such that $L(G') \supseteq L(G)$. G' is called the *skeleton grammar* of G. In order to parse a string according to a VWG,

one first obtains the skeleton grammar and parses the string according to that grammar. If the parse succeeds, it is necessary to examine the parse tree and check that the string is acceptable to the original VWG. The particular hypernotions which have to be checked are determined by the relationship between the original VWG and the skeleton grammar.

For a VWG to be parsable by Wegner's algorithm, it must conform to a number of restrictions, the most onerous of which is that either all hyperrules must be left-bound, or all hyperrules must be right-bound. Mixed non-right-bound and non-left-bound hyperrules are not allowed. Wegner admits that "metaboundedness [i.e. left- or right-boundness] causes TLGs [i.e. VWGs] to degenerate", and he proposes a development, "bracketed grammars" [12], which avoids this problem. Bracketed grammars are not, however, VWGs—they are in fact more like affix grammars—and so they do not fall within the scope of this paper.

The most recent parsing algorithm for pure VWGs is that of Fisher [5]. This algorithm is based on LL(1) parsing techniques. The principal restrictions imposed by the method are that the defining CFG of the left-hand-side hypernotion of each hyperrule must be LL(1) (a slightly stronger condition than that the metasyntax be LL(1)), that the skeleton grammar of the VWG must be LL(1), and that the VWG must be right-bound. The first two restrictions are not onerous, but the algorithm shares with Wegner's algorithm the restriction that the VWG be metabounded, and Wegner's observation, quoted above, applies equally to Fisher's LL(1)-based algorithm.

4. Earley's algorithm applied to VWGs

We consider first some modifications to Earley's context-free parsing algorithm [3] which allow it to work with VWGs.

This is a partial algorithm. It is not of practical use, because it involves the evaluation of a non-computable function μ, and because for certain VWGs and for certain sentences it requires infinite time and space. The algorithm is introduced at this point because it is easier to explain its operation and to argue its partial correctness (i.e. its correctness assuming termination) by relating it to Earley's algorithm, and then to explain how its deficiencies can be overcome, than it would be to present the final algorithm and its explanation in one step. The final (practical) algorithm is presented in Sect. 5 of this paper.

The following notation is used in the algorithm.

Let $G = (M, V, N, T, R_M, R_V, S)$ be a VWG, and let $W = w_1 \ldots w_m$, with $w_i \in T$ for $i = 1, \ldots, m$, be the sentence to be parsed. Set $w_{m+1} = \dashv_T$, a unique stop symbol.

If $p = l : r_1, \ldots, r_n$ is a hyperrule, we denote by $p.l$ the left-hand-side hypernotion l of p, by $p.r_j$ the jth right-hand-side hypernotion r_j of p, and by $|p.r|$ the length n of the right-hand side of p.

Let x_1, x_2 be hypernotions. The Boolean-valued function $\mu(x_1, x_2)$ yields true iff there exists at least one consistent substitution ϕ such that $\phi(x_1) = \phi(x_2)$. Now x_1 and x_2 are defined by CFGs G_1 and G_2 (respectively), whose terminal alphabet is V (the alphabet of small syntactic marks). The function μ is in general not computable, since the question "is $L(G_1) \cap L(G_2) \neq \varnothing$?" is unsolvable for general CFGs G_1, G_2.

There might be many such substitutions ϕ. In the algorithm, Φ_μ denotes the set of all consistent substitutions ϕ determined by the previous evaluation of μ.

The algorithm manipulates objects called *states*. A state is a triple (p, j, b), where

$p = l : r_1, ..., r_n$ is a hyperrule ($n \geq 0$);

j is an integer, $0 \leq j \leq n$, which identifies a position in the right-hand side of the hyperrule p; and

b is an integer, $0 \leq b \leq m+1$, which identifies a position in the sentence being parsed.

Following Earley, a state (p, j, b), where $p = l : r_1, ..., r_n$, will sometimes be written as

$$l : r_1 \ldots r_j \cdot r_{j+1} \ldots r_n$$

where the position of the "dot" in the right-hand side denotes the value of j in the state. Greek letters α, β, γ, δ stand for arbitrary sequences of hypernotions or terminals (or both).

A *state set* is an ordered set of states in which no state occurs more than once. A state set is organized as a first-in-first-out list, so that states are removed from a state set in the same order as that in which they were added.

The algorithm maintains a list of state sets, ss_0, ..., ss_{m+1}, where ss_i contains states which represent the partial parse up to and including the ith symbol of the candidate sentence. The algorithm works from left to right through the sentence, adding states to the state sets as it goes. At the start of the algorithm, ss_0 contains one element, the state (p_0, 0, 0), where p_0 is a hyperrule $\alpha : S$ with the starting symbol S of the VWG on its

right-hand side, and all of the other ss_i (for i = 1, ..., m+1) are empty.

Algorithm ALG_1.

For each i = 0, ..., m, perform the following step.

1. *process states in* ss_i*:*

Process the states of ss_i in order, performing one or more of the following operations to each state $(p, j, b) \in ss_i$.

1.1. if $j < |p.r|$ and $p.r_{j+1}$ is a hypernotion:

predictor:

for each hyperrule $hr = l : r$ such that $\mu(p.r_{j+1}, l)$

for each $\phi \in \Phi_\mu$, add $(\phi(hr), 0, i)$ to ss_i.

1.2. if $j < |p.r|$ and $p.r_{j+1}$ is a hypernotion and $\mu(\lambda, p.r_{j+1})$:

gap scanner:

for each $\phi \in \Phi_\mu$, add $(\phi(p), j+1, b)$ to ss_i.

1.3. if $j < |p.r|$ and $p.r_{j+1}$ is a terminal and $p.r_{j+1} = w_{i+1}$:

terminal scanner:

add $(p, j+1, b)$ to ss_{i+1}.

1.4. if $j \geq |p.r|$:

completer:

for each state $(pb, jb, bb) \in ss_b$

such that $pb.r_{jb+1}$ is a hypernotion and $\mu(p.l, pb.r_{jb+1})$

for each $\phi \in \Phi_\mu$, add $(\phi(pb), jb+1, bb)$ to ss_i.

End of algorithm.

Suppose that ALG_1 terminates when presented with a sentence s and a grammar G. Then the algorithm succeeds, indicating that s belongs to

$L(G)$, iff on termination $(p_0, 1, 0) \in ss_m$.

Explanation.

Suppose for the time being that there are no empty hypernotions, i.e. hypernotions x such that $\mu(\lambda, x)$. Step 1.2 (the gap scanner) will therefore never be applicable. Under this assumption, the algorithm behaves identically to Earley's algorithm; it parses a CFG which is a strict syntax of the VWG. If the VWG does not contain any metanotions, the strict syntax is identical to R_V, $\mu(x_1, x_2)$ yields true iff $x_1 = x_2$, and $\phi \in \Phi_\mu$ is the identity function. Now suppose that the VWG does contain metanotions. Consider the places in the algorithm where a state (p, j, b) is added to a state set. In each such place, p is metanotion-free. In the case of the predictor and the completer, this is so because the value of ϕ is by definition metanotion-free. In the case of the terminal scanner, p is a representation rule, which is by definition metanotion-free. On termination, therefore, the state sets will hold information about the particular context-free strict syntax (or syntaxes), derivable by consistent substitution from the hypersyntax, which generates the sentence.

Now consider the gap scanner. If a possible consistent substitution for a hypernotion x is the empty protonotion, then x is capable of generating the empty terminal word. The gap scanner takes care of this eventuality by adding, for each such state $\alpha : \beta \cdot x \gamma$ (for some α, β, γ), a new state $\alpha : \beta x \cdot \gamma$ in which the dot is advanced past the nullable hypernotion.

5. A new parsing algorithm for VWGs

In this section, we present a new practical parsing algorithm for a large class of VWGs. This algorithm overcomes the deficiencies of the

algorithm presented in the previous section.

It was noted that μ is in general not computable. We observe that μ is however computable if its first argument is a protonotion. In that case, $\mu(pn, hn)$ yields true iff there exists a consistent substitution ϕ such that $\phi(hn) = pn$. This is the CFG parsing problem.

Following Fisher [5], we now define the *defining CFG* $G_D(x)$ of a hypernotion $x \in N$ of G by

$$G_D(x) = (M \cup \{ X_0 \}, V \cup \{ \dashv_V \}, X_0, F \cup \{ X_0 \rightarrow x \dashv_V \})$$

where X_0 is a new metanotion and $\dashv_V$ is a new small syntactic mark. (Note that the definition in [5] is slightly different: it is defined there not for hypernotions, but for hyperrules.)

We can now state the first of the restrictions on the VWG which must be satisfied in order for the parsing algorithm ALG_2 (given later) to work. Suppose that the defining CFG of each hypernotion in the VWG is LL(1). Call this property of the grammar P1. Inspection of the algorithm ALG_2 shows that new hypernotions are created by the parser only by taking old hypernotions and binding their metanotions to protonotions. Therefore, if P1 is true of all hypernotions textually present in the grammar, then all hypernotions manipulated by the algorithm will have LL(1) defining CFGs. The evaluation of $\mu(pn, hn)$ therefore reduces to the LL(1) parsing problem, which is of course efficiently solvable. Furthermore, since LL(1) grammars are unambiguous, if $\mu(pn, hn)$ yields true, then the value of $\phi(m)$ is determined uniquely for all metanotions m which occur in hn. It is not in general the case that ϕ is determined uniquely for metanotions which do not occur in hn.

In fact P1 can be weakened slightly. A left-to-right LL(1) parse binds metanotions unambiguously to protonotions. So, if there are two or more occurrences of a metanotion *m* in a hypernotion *h*, when a protonotion is parsed according to the defining CFG of *h*, the unique binding found for the first occurrence of *m* in *h* fixes the binding for the second and subsequent occurrences as well. For example, given the metasyntax

TALLY1:: TALLY.

TALLY2:: TALLY.

TALLY:: i TALLETY.

TALLETY:: TALLY; EMPTY.

EMPTY:: .

the hypernotion

where TALLY1 less than TALLY1 TALLY2

is acceptable to the algorithm, although its defining CFG is not LL(1). The weaker version of P1 is:

R1. For each hypernotion *h* in the VWG, let *h′* denote the value of *h* after the following operation has been performed: for each metanotion *m* present in *h*, replace the second and subsequent occurrences of *m* in *h′* consistently by some protonotion derivable from *m*. The VWG is acceptable only if the defining CFG of *h′* is LL(1), no matter which substitutions are chosen.

R1 is easily checked. Algorithms for checking the LL(1)-ness of a CFG are well known [6]. The standard algorithms all proceed by constructing "starter" and "follower" sets for non-terminals in the CFG. To check R1, one applies one of the standard algorithms, but neglects to record followers for the second and subsequent occurrences of each metanotion.

We now consider three cases.

Case 1: The grammar is right-bound

Consider first what happens if the VWG is right-bound. Consider the three places in ALG_1 where the function ϕ is applied: in the predictor, in the gap scanner, and in the completer. The argument of ϕ in its application in the gap scanner and in the completer is a hyperrule which is obtained from a state set, i.e. a rule of the strict syntax, which is by definition metanotion-free. In the predictor, ϕ is applied to a hyperrule $hr = l : r$ of the VWG. But the previous evaluation of μ determines a unique substitution for all metanotions m which occur in l. Since the hyperrule is right-bound, all metanotions in r occur also in l. Therefore, for a right-bound VWG, ϕ is determined uniquely.

Case 2: The grammar is left-bound

Now consider how ALG_1 can be modified to work with left-bound grammars. If the VWG is not right-bound, it is no longer true in general that the evaluation of μ in the predictor establishes a unique consistent substitution ϕ: certain metanotions (m_k, say) in r might remain unbound. It is not possible to record all possible consistent substitutions for each m_k if $L(m_k)$ is infinite, since this implies that the set Φ_μ is infinite. To eliminate this problem, we make use of a partial substitution function ϕ_μ, defined as follows:

$$\phi_\mu(v) = v \qquad \text{for } v \in V \text{ or } v = \lambda,$$

$$\phi_\mu(m) = \phi(m) \quad \text{for } m \in M_b,$$

$$\phi_\mu(m) = m \qquad \text{for } m \in M \setminus M_b,$$

where M_b is the set of metanotions for which unique bindings were found

by the previous evaluation of $\mu(pn, hn)$ (i.e. the set of metanotions present in hn), and $\phi(m)$, as defined previously, is the unique binding thus found for the metanotion m.

One result of this modification is that, when a state (p, j, b) is added to a state set, p might not be metanotion-free. Consequently, when μ is evaluated in the predictor and in the completer, its first argument might not be a protonotion. It is necessary to guard against this possibility by changing the conditions attached to steps 1.1 (the predictor) and 1.4 (the completer) in ALG_1 as follows:

1.1. if $j < |p.r|$ and $p.r_{j+1}$ is a protonotion

1.4. if $j \geq |p.r|$ and $p.l$ is a protonotion

Now, if the VWG is right-bound, the operation of the algorithm will not be affected; but if the VWG is not right-bound, the predictor will not be able to proceed when it tries to predict new states from a non-right-bound hyperrule. The algorithm will not in any circumstances produce spurious parses; but it will miss parses in some circumstances, and a means must be found to eliminate this problem.

Algorithm ALG_1 operates top-down. One interpretation of the problem is that while a top-down parse is appropriate for a right-bound VWG, a bottom-up parse is appropriate if the VWG is left-bound. We add a new step 0, *add pre-terminals*, to ALG_1.

0. *add pre-terminals:*

For each hyperrule $hr = l : t \in R_V$ such that $t \in T$ and $t = w_{i+1}$

add $(hr, 0, i)$ to ss_i.

In a bottom-up parse, the first states to be added correspond to the leaves of the parse tree. Step 0 “primes” the state sets, by adding these

initial states, so that a bottom-up parse can proceed. The initial states are derived from the representation rules of the VWG, in an obvious way.

Consider a state (p, 0, b) added by step 0, in which p is necessarily metanotion-free (since representation rules are metanotion-free). The terminal scanner will act on this state, and will create a new state (p, 1, b). This state will now be acted on by the completer. For the algorithm to succeed, the completer must find states (pb, jb, bb) whose right-hand-side hypernotions $pb.r_{jb+1}$ "match" the protonotion $p.l$. In Earley's standard algorithm, these states would have been entered previously, by the (top-down) predictor. Since the VWG algorithm works bottom-up in the left-bound case, these states have not yet been added. They must be entered by some other means. This is the function of the *bottom-up predictor*, which is called immediately before the completer, when the dot in a state has progressed as far as the end of the right-hand side; i.e. it acts on states $x : \alpha \cdot$.

1.3*a*. if $j \geq |p.r|$ and $p.l$ is a protonotion:

bottom-up predictor:

for each hyperrule $hr = l : r_1, ..., r_n$ such that $n \geq 1$ and

r_1 is a hypernotion and $\mu(p.l, r_1)$

add $(\phi_\mu(hr), 0, b)$ to ss_b.

The bottom-up predictor adds a new state for each hyperrule hr in the VWG whose leading right-hand-side hypernotion $hr.r_1$ "matches" the left-hand side of the hyperrule p under consideration. The new state $\beta : \cdot x \gamma$ (say) is added to ss_b, since at this point the parser has just scanned the whole of an x, starting at position b in the sentence. This new state represents a provisional state of the parse at the next level up in the parse

tree, nearer the root. After the bottom-up predictor has entered these new states in ss_b, the completer will be called. It will act on all appropriate states in ss_b. Some of these will have been entered by the top-down predictor, others by the bottom-up predictor. When the dot in the new state has reached the end of the right-hand side, the bottom-up predictor will again be called, and will add a yet higher node to the parse tree. Eventually, a complete "frontier" from the terminal up to the root will have been created. The bottom-up predictor performs the same function for left-bound hyperrules as the top-down predictor performs for right-bound hyperrules.

It might at first be thought that this behaviour is correct only if the empty terminal word is not derivable from r_1 (after consistent substitution). Suppose the bottom-up predictor adds the state $\beta : \cdot\ x\ \gamma$, and there is a consistent substitution ϕ such that $\phi(x) \Rightarrow_G^* \lambda$. Ought not the bottom-up predictor to add a state $\beta : x \cdot \gamma$ as well? In fact this is not necessary, since this extra state will be added later by the gap scanner.

Now consider the action of the completer in the left-bound case. Since the hyperrule *pb* is left-bound, all metanotions in *pb.l* also occur somewhere in *pb.r*. They might not all be present in $pb.r_{jb+1}$, so the value of $\phi_\mu(pb)$ will not necessarily be metanotion-free. However, by the time the dot has reached the end of the rule, i.e. when the new state entered by the completer has the form $\alpha : \beta \cdot$, every metanotion present in *pb.l* will have been substituted for, so *pb* in this final state will be metanotion-free. Note that it is this dot-final state which is acted on by the bottom-up predictor. It is therefore not necessary (as might at first be thought) for every hyperrule $hr = l : r_1, ..., r_n$ to be left-bound in its first component only, by

which is meant that every metanotion in l occurs also in r_1. It is necessary only that every hyperrule be left-bound.

It is necessary to show that the addition of steps 0 and 1.3*a* do not invalidate the operation of the algorithm in the right-bound case. This is so because step 0 adds states corresponding to representation rules, which would be added eventually by the top-down predictor in the right-bound case. Step 1.3*a* adds states which differ from those added by the top-down predictor only in being (possibly) less fully instantiated. No spurious parses can be generated; but the fact that states, differing only in their degree of instantiation, can be added in different ways, means that care must be taken to eliminate spurious ambiguity in the form of multiple identical parse trees. This point will be considered again later.

Case 3: The grammar is non-right-bound and non-left-bound

Finally, consider what happens if the VWG contains both non-left-bound and non-right-bound hyperrules. Consider a frontier f_0, ..., f_k of the parse tree, where f_0 is the root and f_k is a leaf. For any such frontier, there will exist i and j such that the top part f_0, ..., f_i will be derivable top-down from right-bound hyperrules, and the bottom part f_j, ..., f_k will be derivable bottom-up from left-bound hyperrules. The grammar is acceptable to the algorithm iff, for all frontiers of all parse trees, $i \geq j$.

It is unfortunate that this is a restriction on parse trees. What one would like is a restriction on grammars. A sufficient (but not necessary) condition on the grammar is the following. Define the *cross-reference set* $X(G)$ of a VWG $G = (M, V, N, T, R_M, R_V, S)$ by

$X(G) = \{ (h_R, h_L) \mid h_R, h_L \in N,$

h_R occurs on the right-hand side of some hyperrule $\in R_V$,

h_L occurs on the left-hand side of some hyperrule $\in R_V$,

$\mu(h_R, h_L) \}$.

A hyperrule $hr \in R_V$ is

of the *type L* iff it is left-bound but not right-bound;

of the *type R* iff it is right-bound but not left-bound;

of the *type LR* iff it is both left-bound and right-bound;

of the *type X* iff it is neither left-bound nor right-bound.

We require that:

R2. The hypersyntax must not contain any type X rules.

R3. There must be no ordered pairs $(h_R, h_L) \in X^+(G)$ where h_R occurs on the right-hand side of a type L rule and h_L occurs on the left-hand side of a type R rule, *unless* all metanotions in h_R occur also in preceding hypernotions in the right-hand side of the type L rule.

If the grammar contains a hyperrule which is neither left-bound nor right-bound, it can take part neither in a top-down parse nor in a bottom-up parse. It is therefore reasonable to forbid such hyperrules (R2). The classification of the remaining rules as type R, L or LR induces a partition of the hyperrule set into rules which may be applied in the “top” portion of the parse tree, or the “bottom” portion, or both (respectively). Ignoring for the moment the “unless” clause, R3 prohibits a cross-reference from the right-hand side of a “bottom”-applicable rule to the left-hand side of a “top”-applicable rule. It is easy to see that all parse trees resulting from a grammar which conforms to R2 and R3 will conform to the two-part

frontier restriction.

The "unless" clause is intended mainly to allow hyperrules with empty right-hand sides in grammars which are mainly left-bound. Such hyperrules are very common in practical grammars, e.g. in the grammar of the Report. Wegner [11] calls them ε-*hyperrules*. ε-hyperrules are either metanotion-free (and therefore type LR), or type R. But an ε-hyperrule can be applied only at the bottom of the parse tree, which is precisely where type R rules are not allowed. If all metanotions in h_R occur also in preceding hypernotions, however, the left-to-right parse (specifically, the completer) will have fixed bindings for all of these metanotions by the time h_R is encountered, so a top-down parse can proceed.

$\mu(x_1, x_2)$ is not computable if x_1 and x_2 are hypernotions, so R3 is not checkable. We can however define a superset matching function μ' such that $\mu(x_1, x_2) \Rightarrow \mu'(x_1, x_2)$, and use μ' instead of μ when checking R3. If the amended R3 (using the superset μ' function) is satisfied for a grammar, then the original R3 will be satisfied (but the implication is not of course both-ways). μ' can be defined in many different ways, but the following recursive algorithmic definition has been found useful:

$\mu'(x_1, x_2)$ =

if $x_1 \in V^*$ then $\mu(x_1, x_2)$

else if $x_2 \in V^*$ then $\mu(x_2, x_1)$

else if $x_1 = \alpha y_1$ and $x_2 = \beta y_2$ for some $\alpha, \beta \in V$

then $(\alpha = \beta) \wedge \mu'(y_1, y_2)$

else if $x_1 = y_1\alpha$ and $x_2 = y_2\beta$ for some $\alpha, \beta \in V$

then $(\alpha = \beta) \wedge \mu'(y_1, y_2)$

else true.

Termination

It is necessary to show that the algorithm terminates, i.e. that it is not possible for it to keep adding new states to state sets indefinitely. To show this, we consider the six places in ALG_2 where a new state is added to a state set.

Clearly step 0 cannot cause problems, because the total number of states added is at most m+1 times the number of hyperrules in the grammar, which is finite. The gap scanner, the terminal scanner and the completer cannot cause problems either, because they act on a state $\alpha : \beta \cdot x\ \gamma$ to yield a new state $\alpha : \beta\ x \cdot \gamma$ in which the dot has advanced one position. An infinite sequence of these operations cannot occur, since the dot would then "fall off the end of the rule".

It is, however, possible for an infinite sequence of calls of the bottom-up and top-down predictor to be made (in any order). Since there are only finitely many hyperrules, this can happen only if a hyperrule *hr* is acted on more than once by the top-down or bottom-up predictor, without making progress through the sentence. Such an infinite loop can occur only if the strict syntax is left-recursive. We now establish a restriction on the VWG which guarantees that no strict syntax will be left-recursive.

Define the *initial cross-reference set* $Y(G)$ of a VWG $G = (M, V, N, T, R_M, R_V, S)$ by

$Y(G) = \{ (h_R, h_L) \mid h_R, h_L \in N,$

h_R occurs on the right-hand side of some hyperrule

$hr = r_1, ..., r_n \in R_V,$

h_L occurs on the left-hand side of some hyperrule $\in R_V$,

$\mu(h_R, h_L)$,

$\exists$ an integer k, $1 \leq k \leq n$, and a consistent substitution ϕ

such that $\phi(r_i) \Rightarrow_G^* \lambda$ for each $i = 1, ..., k-1$, and $h_R = r_k$ }.

Informally, $Y(G)$ is the set of pairs (h_R, h_L) such that h_R occurs in a leading position on the right-hand side of a hyperrule, and h_L occurs on the left-hand side of a hyperrule, and h_R "matches" h_L; but taking into account the fact that hypernotions which precede h_R might, after consistent substitution, derive the empty terminal word. Now this function is not computable, for the same reason that $X(G)$, considered earlier, is not computable; but by the same token, it is possible to compute a "superset initial cross-reference" which suffices for the purpose of checking the grammar.

We can now define a VWG G to be *left-recursive* iff $\exists$ a hyperrule $hr = l : r \in R_V$ such that $(l, l) \in Y^+(G)$. The algorithm ALG_2 will terminate if (but not only if) condition R4 is satisfied:

R4. The VWG must not be left-recursive.

Despite the apparent complexity of the definition of $Y(G)$, R4 is in practice often easily decidable by inspection of the VWG. It is not particularly onerous in practice. It is a rather strong condition, in the sense that the algorithm will often terminate on grammars for which R4 does not hold. Because there are only finitely many different hypernotions of a given length, non-termination implies that a left-recursive cycle of

applications of the bottom-up or top-down predictor causes the length of a particular hypernotion to increase without limit. This may be considered as a property of the grammar, not of the algorithm. A VWG which has the property that a partial parse—even of a parse which will eventually fail—involves the elaboration of hypernotions of infinite length may fairly be said to be ill-behaved. Furthermore, this property can usually be checked by (human) inspection. Consequently, the author's implementation of the algorithm checks R4, but issues a "warning" message rather than a "failure" message if it is not met. "Failure" messages are issued if any of the other conditions R1–R3 is not met. (Note that it is not correct to say that a failure message is issued if *and only if* any of R1–R3 is not met, since R3 is not decidable.)

Algorithm ALG_2.

For each $i = 0, ..., m$, perform the following steps.

0. *add pre-terminals:*

For each hyperrule $hr = l : t \in R_V$ such that $t \in T$ and $t = w_{i+1}$

add $(hr, 0, i)$ to ss_i.

1. *process states in* ss_i*:*

Process the states of ss_i in order, performing one or more of the following operations to each state $(p, j, b) \in ss_i$.

1.1. if $j < |p.r|$ and $p.r_{j+1}$ is a protonotion:

top-down predictor:

for each hyperrule $hr = l : r$ such that $\mu(p.r_{j+1}, l)$

add $(\phi_\mu(hr), 0, i)$ to ss_i.

1.2. if $j < |p.r|$ and $p.r_{j+1}$ is a hypernotion and $\mu(\lambda, p.r_{j+1})$:

gap scanner:

add $(\phi_\mu(p), j+1, b)$ to ss_i.

1.3. if $j < |p.r|$ and $p.r_{j+1}$ is a terminal and $p.r_{j+1} = w_{i+1}$:

terminal scanner:

add $(p, j+1, b)$ to ss_{i+1}.

1.3*a*. if $j \geq |p.r|$ and $p.l$ is a protonotion:

bottom-up predictor:

for each hyperrule $hr = l : r_1, ..., r_n$ such that $n \geq 1$ and

r_1 is a hypernotion and $\mu(p.l, r_1)$

add $(\phi_\mu(hr), 0, b)$ to ss_b.

1.4. if $j \geq |p.r|$ and $p.l$ is a protonotion:

completer:

for each state $(pb, jb, bb) \in ss_b$

such that $pb.r_{jb+1}$ is a hypernotion and $\mu(p.l, pb.r_{jb+1})$

add $(\phi_\mu(pb), jb+1, bb)$ to ss_i.

End of algorithm.

6. Construction of the parse tree

A minor modification to ALG_2 allows a parse tree to be deduced from information in the state sets at the conclusion of a successful parse. Two relations are defined, *rlink* and *uplink*, which are initially empty.

1. Whenever the gap scanner or terminal scanner acts on a state $q_1 = \alpha : \beta \cdot x\ \gamma$ to produce a new state $q_2 = \alpha : \beta\ x \cdot \gamma$, the pair (q_1, q_2) is added to rlink.
2. Whenever the completer acts on a state $q_1 = x : \alpha \cdot$ to complete a state $q_2 = \beta : \gamma \cdot x\ \delta$ yielding a new state $q_3 = \beta : \gamma\ x \cdot \delta$, the pair (q_2, q_3) is added to rlink, and also the pair (q_3, q_1) is added to uplink.

From this information, a factorized parse tree can be readily constructed. Strictly speaking, this is true only if the grammar has at most a finite degree of ambiguity: if that is not the case, the parse "tree" will be a cyclic graph (but it can still be constructed). The technique is similar to that used in many implementations of Earley's CFG parsing algorithm.

Since the parse tree is meant to represent a strict syntax, each component in the tree must be either a protonotion or a terminal. Consider the three places where an entry (q_1, q_2) is added to the rlink relation: in

the gap scanner, the terminal scanner, and the completer. In each case, if $q_1 = \alpha : \beta \cdot x\ \gamma$ and $q_2 = \alpha : \beta\ x' \cdot \gamma$, x' is metanotion-free (but x might not be). In the case of the terminal scanner, x' is a terminal. In the other cases, consistent substitution has just been applied and all metanotions in x have been substituted for, so x' is a protonotion. So, whenever an entry (q_1, q_2) is added to the rlink relation, whether by the gap scanner, the terminal scanner or the completer, if $q_2 = \alpha : \beta\ x' \cdot \gamma$, then the x' before the dot is either a terminal or a protonotion. Other components are not guaranteed to be metanotion-free, and cannot in general be used to construct elements of the parse tree.

It was noted in Sect. 5 that a state set might contain two states (p_1, j, b) and (p_2, j, b) which are identical except that p_1 is more fully instantiated than p_2. This would give rise to multiple spurious ambiguity in the final parse tree, if special action were not taken. The "extra" state (p_2, j, b) cannot simply not be added to the state set, since it might well be needed to allow a genuine alternative parse to proceed.

A better approach is to delete spurious ambiguity while the parse tree is being constructed. It is assumed that the factorized parse tree will contain "alternative" nodes whose sub-trees are alternatives which arise from ambiguity in the strict syntax, either genuine or spurious. It is easy to detect the fact that two sub-trees of an "alternative" node are identical, and to retain only one them. This process is made efficient by adopting a well-known technique from the practice of compiler construction, and always storing identical sub-trees at identical addresses in computer memory.

7. Time and space bounds

The following parameters are relevant to a consideration of time bounds for ALG_2:

G, the number of hyperrules in the VWG;

n, the length of the sentence being parsed.

If the VWG is free of metanotions, the algorithm works in time $O(G^2n^3)$, for the following reasons. In the absence of metanotions, the evaluation of μ and ϕ takes time independent of G and n. Earley's CFG parsing algorithm, as is well known, works in time $O(N^2n^3)$, where N is the number of rules in the CFG. In the present algorithm, N is the number of rules in the strict syntax, which is bounded by G. Earley's proof of the time complexity of his algorithm [3] is not affected by the presence of the bottom-up predictor, which takes time $O(Nn)$, the same as the top-down predictor. Step 0 takes time $O(Gn)$, so does not worsen the overall time bound. The algorithm therefore takes time $O(G^2n^3)$.

The situation is of course worse if the grammar contains metanotions. There are grammars on which the algorithm takes exponential time (as a function of n). Consider, for example, the following grammar, taken from Fisher [5].

B:: b BETY.

BETY:: B; EMPTY.

EMPTY:: .

start: a symbol, b.

B: a symbol, BB; b symbol.

This grammar satisfies R1–R4 and generates the language $\{ a^n b \mid n \geq 1 \}$. However, as Wegner points out (for he has a similar

example), the number of *b*'s in the hypernotion BB doubles at each derivation step, and is therefore exponentially related to the length of the input word. The evaluation of μ(*pn*, *hn*), in those cases in which it yields true, takes time which is linearly related to the length of the protonotion *pn*. Therefore, for this grammar the evaluation of μ can take time $O(e^n)$, and so the algorithm as a whole takes time $O(e^n)$. (Since we are considering a specific grammar, it is not appropriate to express the bound as a function of *G*.)

We take the view that this exponential behaviour is exceptional. Speaking informally, it is clear that in this example there is "no need" for the exponential growth of protonotions. The grammar would generate exactly the same language if BB in the last hyperrule were replaced by B; the maximum length of a protonotion would then be bounded by a constant, and the algorithm would take time $O(n^3)$. Further support for this view is provided by a wide variety of VWGs, including both of the example grammars EX1 and EX2 (Figs. 1 and 2). In these grammars, it is easy to see that both the maximum length *p* of a protonotion and the maximum size *r* of the strict syntax (i.e. the number of rules at the conclusion of the parse) grow no faster than linearly with *n*. We call grammars with this property *linearly frugal*. Similarly, if *p* and *r* are both $O(f(n))$ for some function *f*, we say that the grammar is *f*(*n*)*-frugal*.

For an *f*(*n*)-frugal grammar, the size of the strict syntax grows as *f*(*n*) as the parse progresses. We therefore replace *N* by *Gf*(*n*) in Earley's bound of $O(N^2n^3)$, giving a bound of $O(G^2n^3f^2(n))$ on the number of evaluations of μ and ϕ. Since the LL(1) parsing of a protonotion with respect to a hypernotion takes time $O(k)$, where *k* is the length of the protonotion, and

since by definition $k = O(f(n))$ for $f(n)$-frugal grammars, the evaluations of μ and ϕ are $O(f(n))$ operations. The overall time bound is therefore $O(G^2n^3f^3(n))$. In the special case of a linearly-frugal grammar, this simplifies to $O(G^2n^6)$.

Inspection of the large grammar of the Report leads one to believe that the Algol 68 grammar, though not linearly frugal, is probably n^2-frugal in the worst case, or n log n-frugal for typical sentences (programs). This is because extensive use is made of the ability of protonotions to "bundle up" context information such as a list of declarations in scope. This information is bounded by the length of the program and typically proportional to the depth of bracket nesting in the program. The fact that such a large grammar, designed without thought for its suitability for algorithmic parsing, is polynomial-frugal gives one confidence that the algorithm just described would work in polynomial time in a large number of practical cases.

Similar considerations apply to the space bound of the algorithm. The main data structures are the lists of state sets and the strict syntax. Examples can be found which cause the length of protonotions, and hence the space needed to store the states and the strict syntax, to grow exponentially with the length of the input, but one might hope for polynomial behaviour for a large class of grammars.

Deussen [1] defines *type L* and *type R grammars*. A VWG is of type L if all its hyperrules are left-bound, and of type R if all its hyperrules are right-bound (but there are other restrictions). Deussen and Mehlhorn [2] showed that a language can be generated by a type L grammar if and only if it can be generated by a type R grammar, and that the set of languages

which can be so generated is EXSPACE, the class of languages which are parsable in exponential space. Since the restrictions R1–R4 permit a mixture of non-left-bound and non-right-bound rules, it is interesting to speculate whether there exists a language which lies outside EXSPACE but possesses a grammar which satisfies R1–R4, and which is therefore acceptable to the algorithm. The answer to this question is not known at present.

A proper treatment of time and space bounds, an investigation of the relationship between the class of languages parsable by the algorithm and the class EXSPACE, and the formulation of conditions for ensuring polynomial behaviour, will be topics for future research.

8. Some examples

Figs. 1 and 2 show example VWGs which are parsable by the algorithm. Each hyperrule is annotated with its type (L, R or LR).

Grammar EX1 (Fig. 1) generates the language $\{ a^n\ b^n\ c^n \mid n \geq 1 \}$. This grammar is left-bound. The last three rules (*d–f*) are representation rules. It is interesting to note that if rule *b* is changed to

i TALLY LETTER s: TALLY LETTER s, i LETTER s.

the grammar becomes left recursive, but the parsing algorithm still terminates. This illustrates the fact that R4 is rather too strong. Note also that if rule *b* is changed to

TALLY i LETTER s: i LETTER s, TALLY LETTER s.

the grammar is no longer acceptable, since R1 has been violated.

Grammar EX2 (Fig. 2) is a larger example. The grammar generates a skeleton language of applied and defining occurrences. This example is

intended to mirror the rule in a typical programming language that each applied occurrence of an identifier (e.g. an assignment to a variable, or a procedure call) must be preceded by a defining occurrence of that identifier (e.g. a variable declaration, or a procedure declaration). In the example, an applied occurrence of a variable is represented by the terminal string **A** *name* and a defining occurrence by **D** *name*, where *name* is the name of the variable. Applied occurrences may be followed by assignments which are represented by = **V**. For example, some legal terminal strings are:

D carol **D** mary **A** carol **D** beth

D jane **D** susan **D** jane **A** susan = **V**

whereas

D june **A** april

is illegal. The example is adapted from Pagan [7].

We adopt the usual convention that, for each metanotion *m* defined in a grammar, the ten metanotions *md* where *d* is a digit are defined implicitly by

md :: *m*.

So, for example, the metanotion TAG1 which is used in the grammar is defined implicitly by

TAG1:: TAG.

A TAG is an identifier. A TAGS is a list of TAGs. The informal meaning of "TAGS statement sequence" is "a statement sequence which contains defining occurrences of the named TAGS". The predicate (the hypernotion which starts with "where") in rule *c* enforces the condition that an applied occurrence of a TAG must be preceded by a defining occurrence for the same TAG. The obvious way to define this predicate is

where TAG is in TAGSETY1 TAG TAGSETY2: EMPTY.

but this hypernotion does not have an LL(1) defining CFG. Instead, the grammar uses the two hyperrules *d* and *e*. The logic behind the definition is that a TAG1 is present in a TAG2 TAGSETY if the TAG1 is the same as the TAG2 (i.e. it comes first in the list of TAGs), or (recursively) if the TAG1 is present in the TAGSETY.

Hyperrules *i* and *j* constitute what the Report calls a *general hyperrule*. A NOTION matches any non-empty protonotion which does not contain angle brackets; so for example a "<TAG assignment> option" is either a "TAG assignment" or the empty terminal word, and in general an "<*x*> option" is either an *x* or the empty terminal word. The angle brackets (which are letters of *V*) in rules *h*–*j* are necessary. If they were omitted, the hypernotion "NOTION option" would not have an LL(1) defining CFG.

Hyperrules *d* and *j* are, strictly speaking, type X and not type R, since the metanotion EMPTY occurs on their right-hand sides but not on their left-hand sides. The only protonotion derivable from EMPTY is, however, the empty protonotion; and if the empty protonotion is substituted for EMPTY in the hyperrules, the rules become type R. This is a trivial matter of notation.

Finally, note that rule *e* (for example) is type R, and there is a cross-reference from the right-hand side of a type L rule (rule *c*) to the left-hand side of rule *e*. R3 is satisfied, however, since both metanotions in the hypernotion "where TAG is in TAGS" in rule *c* have occurred previously in the right-hand side of the rule.

Fig. 3 shows the parse tree which results from the parsing of the sentence **D** *a b* **A** *a b*.

9. Conclusions

An algorithm has been described which efficiently parses a large class of van Wijngaarden grammars. Restrictions on the grammar have been derived and stated which guarantee parsability. None of these restrictions is particularly onerous or difficult to comply with. One of the restrictions (R4) is rather too strong: the algorithm will parse many grammars which do not conform to this restriction.

Previous algorithms have suffered from the deficiency that they work only with grammars which are left- or right-bound. The new algorithm does not have this restriction. It is believed that the algorithm is sufficiently general that the elegance, conciseness, expressive power and ease of use of van Wijngaarden grammars will now be able to be exploited in real practical applications.

The algorithm has been implemented in the programming language C, and a practical parser has been constructed. The examples in Figs. 1–3 were prepared using this implementation.

10. Summary of restrictions

R1. For each hypernotion h in the VWG, let h' denote the value of h after the following operation has been performed: for each metanotion m present in h, replace the second and subsequent occurrences of m in h' consistently by some protonotion derivable from m. The VWG is acceptable only if the defining CFG of h' is LL(1), no matter which substitutions are chosen.

R2. The hypersyntax must not contain any type X rules.

R3. There must be no ordered pairs $(h_R, h_L) \in X^+(G)$ where h_R occurs on

the right-hand side of a type L rule and h_L occurs on the left-hand side of a type R rule, *unless* all metanotions in h_R occur also in preceding hypernotions in the right-hand side of the type L rule.

R4. The VWG must not be left-recursive.

As mentioned in Sect. 2, it is also required that, for all derivations of finite sentences, the strict syntax is finite. This can properly be considered a "well-formedness" property for VWGs in general, rather than a restriction of the parsing method.

Acknowledgement

It is a pleasure to acknowledge the contribution of one anonymous referee, whose helpful suggestions have improved the paper.

References

1. Deussen, P: A decidability criterion for van Wijngaarden grammars. Acta Informat. **5**, 353–375 (1975)

2. Deussen, P. and K. Mehlhorn: Van Wijngaarden grammars and space complexity class EXSPACE. Acta Informat. **8**, 193–199 (1977)

3. Earley, J: An efficient context-free parsing algorithm. CACM **13(2)**, 94–102 (February 1970)

4. Fisher, A.J.: The generation of parsers for two-level grammars. Thesis, Prifysgol Cymru, Aberystwyth, Wales (1982)

5. Fisher, A.J.: Practical LL(1)-based parsing of van Wijngaarden grammars. Acta Informat. **21**, 559–584 (1985)

6. Lewis, P.M. II, D.J. Rosencrantz and R.E. Stearns: Compiler design theory. Reading, Massachusetts: Addison-Wesley, 1976

7. Pagan, F.G.: Formal specification of programming languages. Englewood Cliffs, New Jersey: Prentice-Hall International, 1981

8. Salomaa, A.: Formal languages. London: Academic Press, 1973

9. Simonet, M.: W. grammaires et logique du premier ordre pour la définition et l'implantation des langages. Thesis, l'Université Scientifique et Médicale de Grenoble (July 1981)

10. Sintzoff, M.: Existence of a van Wijngaarden syntax for every recursively enumerable set. Annales de la Société Scientifique de Bruxelles, T. 81, II (1967)

11. Wegner, L.M.: On parsing two-level grammars. Acta Informat. **14**, 175–193 (1980)

12. Wegner, L.M.: Bracketed two-level grammars—a decidable and practical approach to language definitions. ICALP 79, Graz. Lecture Notes in Computer Science **71**, p. 668. Berlin: Springer-Verlag, 1979

13. Wijngaarden, A. van *et al.* (eds.): Revised report on the algorithmic language Algol 68. Berlin: Springer-Verlag, 1976